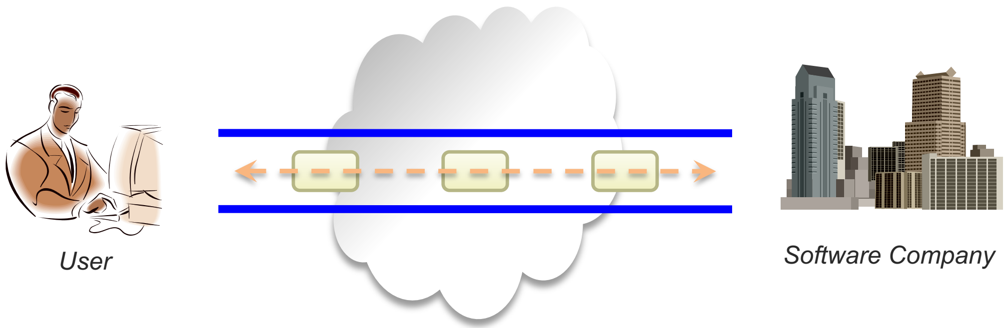
Message Authentication Codes

Jong Hwan Park

Message Authentication

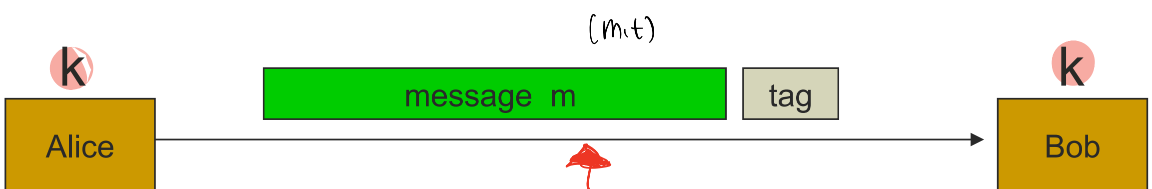
- Confidentiality vs. message integrity
  비밀성 메시지 무결성
  - Distributing patch files or updated programs from a server
  - Not necessary to hide content



- Message integrity is necessary to ensure that: 2가지 목적.
  - (a) The message is sent from the very company 메시지가 송신자로부터 왔다 (Sender Auth)
  - (b) The message is not changed during transmission 메시지가 전송중 위조변조X (message Auth)
- Encryption is sufficient to provide message integrity?
  - Changing the amount by flipping a bit: 000001$ → 100001$

Message Authentication Codes (MAC) (1)

- Definition of MAC = (Gen, Mac, Verify) 대칭키환경.
  - Gen(1^n) → a private-key k
  - Mac(k, m) → a tag t
  - Verify(k, m, t) → output 1 *if m is valid*, otherwise 0



{(mi, ti)}를 알고 있다
k없이 정상검증 되는 (m*, t*)를 생성

Generate tag:
tag ← Mac(k, m)

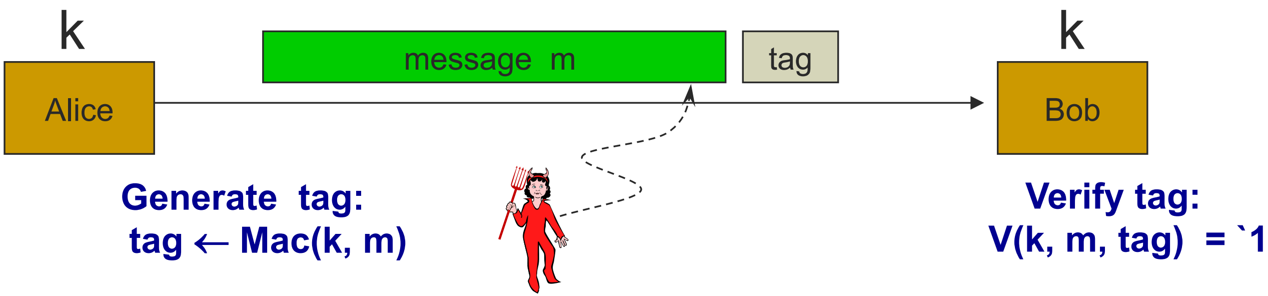
Verify tag:
V(k, m, tag) = '1' 는 원본
0 는 수정

나와 동일한 k를 가진 이가 보내야만 내가 검증을 정상적으로 통과 (Sender 인증)
mac 검증을 통해 메시지가 변했는지 확인가능.

- A private-key k should be shared in advance
  - Key sharing problem occurs as in symmetric-key encryption
- (m, t) is transmitted 숨길필요X
  - The message m is revealed to anyone

Message Authentication Codes (MAC) (2)

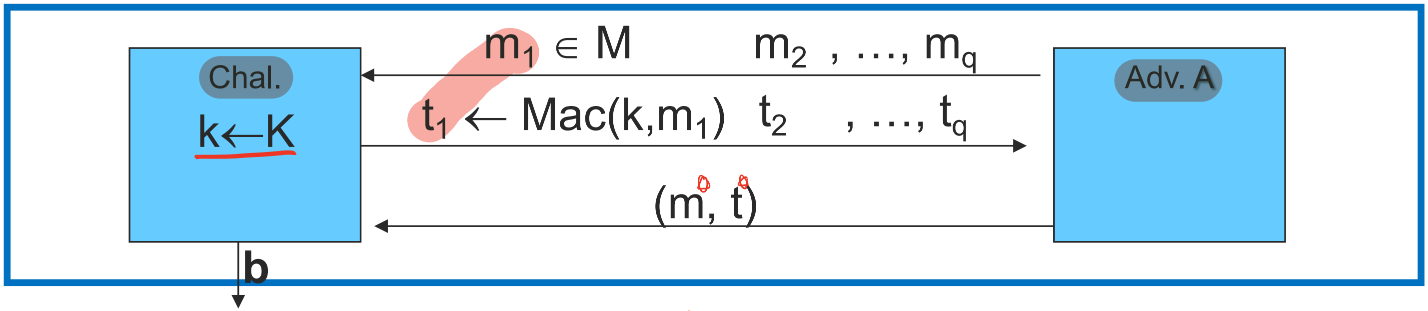
- Security of MAC = (Gen, Mac, Verify) MAC에대한 chose message attack
  - Allow adversary to request MAC tags $\{t_i\}$ for *any* message $\{m_i\}$ Chosen massage (m_i, t_i)를 원하는 수동적으로 얻으면 원하는 m_i에대한 t생성
  - Ensure that no PPT adversary is able to generate $(\tilde{m}, \tilde{t})$ such that
    - (1) $\tilde{t}$ is valid on $\tilde{m}$, i.e., Verify(k, $\tilde{m}$, $\tilde{t}$) =1
    - (2) $\{\tilde{m}, \tilde{t}\} \notin \{(m_1, t_1), \cdots, (m_q, t_q)\}$ ← Not even a previous message m_i



  - The output $\tilde{m}$ is not necessarily meaningful (why ?): existential forgery 존재하면 공격 성공
  - MAC does not offer protection against '*reply attacks*'
    - Two common technique: use of sequence numbers or time-stamps

Modeling Security of MACs

- For a MAC I=(Mac,V) and A, define a MAC security game as:



$$\begin{cases} \mathbf{b}=1 & \text{if } V(k,m,t) = 1 \text{ and } (m,t) \notin \{ (m_1,t_1), \ldots, (m_q,t_q) \} \\ \mathbf{b}=0 & \text{otherwise} \end{cases}$$

- Def: I=(Mac,V) is a **secure MAC** if for all "efficient" A:

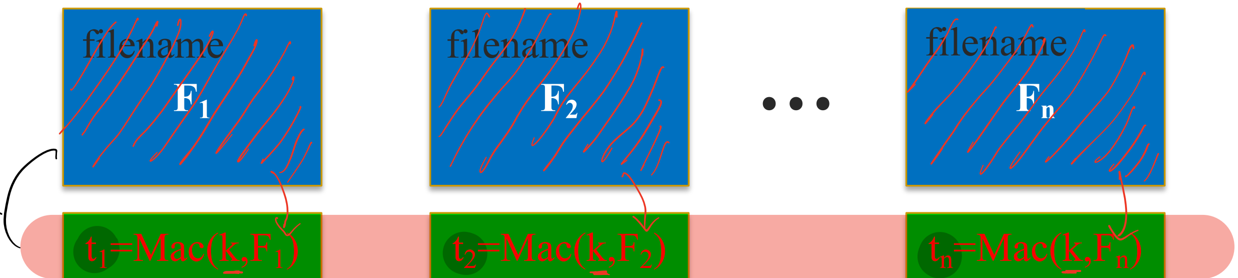
  $\Pr[A_{MAC} \text{ wins}] = \Pr[\text{Chal. outputs } 1]$ is "negligible."

  = Secure MAC

Example: protecting system files

1. hash-mac : hash함수를 이용한 mac
2. C-mac : 대칭키암호 CBC 이용.

- Suppose at install time the system computes:



k derived from user's password

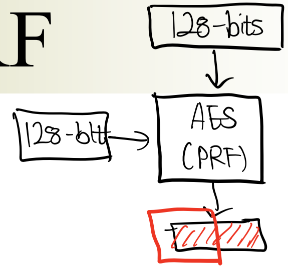
h(pw) = k

비교하는것이 MAC 값들.
사용자의 pw로부터 도출된 key가 필요/ 대칭키방식 아 전자서명으로 가능.

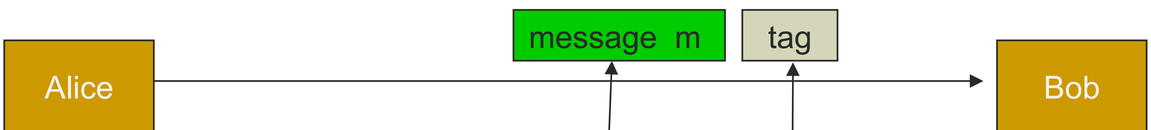
안전한 저장영역 : Virus가 침투해도 바꿀수없는영역.
= Secure HW Storage

- Later a virus infects system and modifies system files
- User reboots into clean OS and supplies his password
  - Then: secure MAC ⇒ all modified files will be detected

MAC construction from PRF



- MAC for fixed-length messages:
- For a PRF **F: K × X ⟶ Y** define a MAC I_F = (Mac, V) as:
  - Mac(k, m) := F(k, m) ≃ Truncated (잘려진 길이로 MAC tag)
    AES
  - V(k, m, t): output `1′ if t = F(k, m) and `0′ otherwise



tag ← F(k,m)

128-bit for AES

|Y| should be large

≃ 96 bit

accept msg if

tag = F(k,m)

복호화 필요X = PRF처럼 한쪽 방향으로만 구하고 비교해주면 된다.
너무 짧으면 X. (확률공격을 막을수 있는 범위에서)

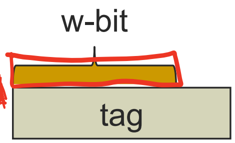
- Thm:

If **F: K×X⟶Y** is a secure PRF and 1/|Y| is negligible (i.e. |Y| is large) then I_F is a secure MAC for fixed-length messages

In Practice

- AES: a MAC for 16-byte messages

- Main question: how to convert Small-MAC into a Big-MAC?
  - Two main constructions used in practice:
    - **CBC-MAC** (banking – ANSI X9.9, X9.19, FIPS 186-3)
    - **HMAC** (Internet protocols: SSL, IPsec, SSH, …)
  - Both convert a small-PRF into a big-PRF



- Truncating MACs based on PRFs

If (Mac,V) is a MAC based on a secure PRF outputting n-bit tags,

the truncated MAC outputting w bits is secure

… as long as $1/2^w$ is still negligible (say $w \geq 64$)

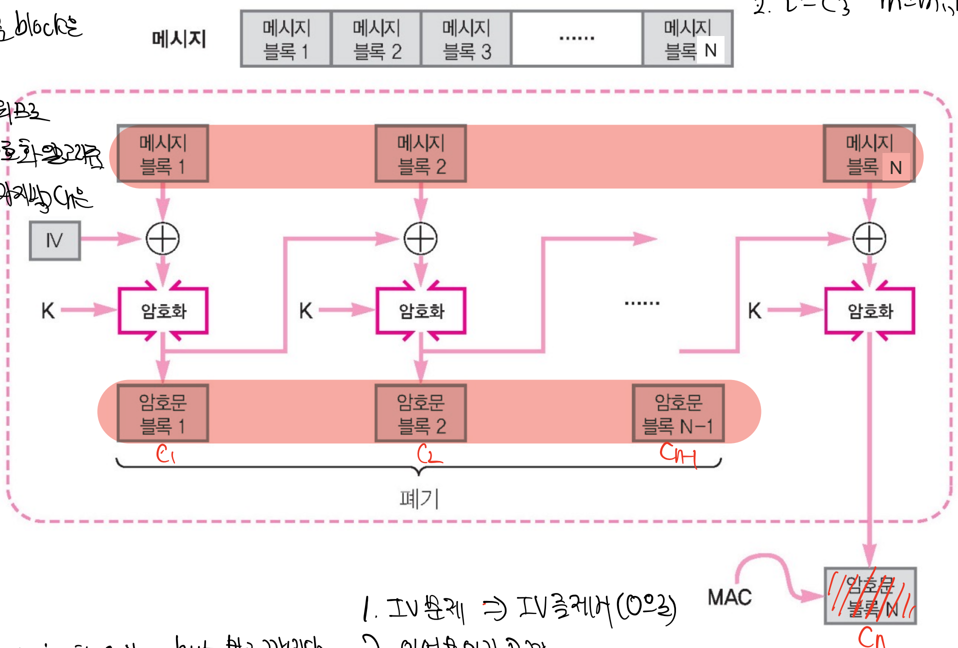
CBC-MAC – simple idea

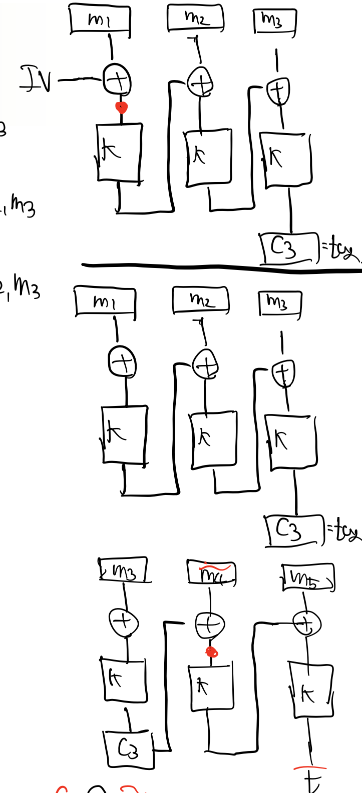
- A Naïve approach using CBC mode

1. (IV, tag), $m = m_1, m_2, m_3$
$\widetilde{IV} \oplus \widetilde{m_1} = IV \oplus m_1$
$\Rightarrow (\widetilde{IV}, tag) = t^{\oplus}$, $m^{\oplus} = \widetilde{m_1}, m_2, m_3$

2. $t = C_3$ $m = m_1, m_2, m_3$

1bit라도 변경되면 마지막 암호문 block은 원본과 다르다.
⇒ 평문에서의 error가 파급되므로 1bit라도 평문이 달라지면 암호화 알고리즘을 거치면서 bit가 바뀌게되고 마지막 Cn은 바뀐다.





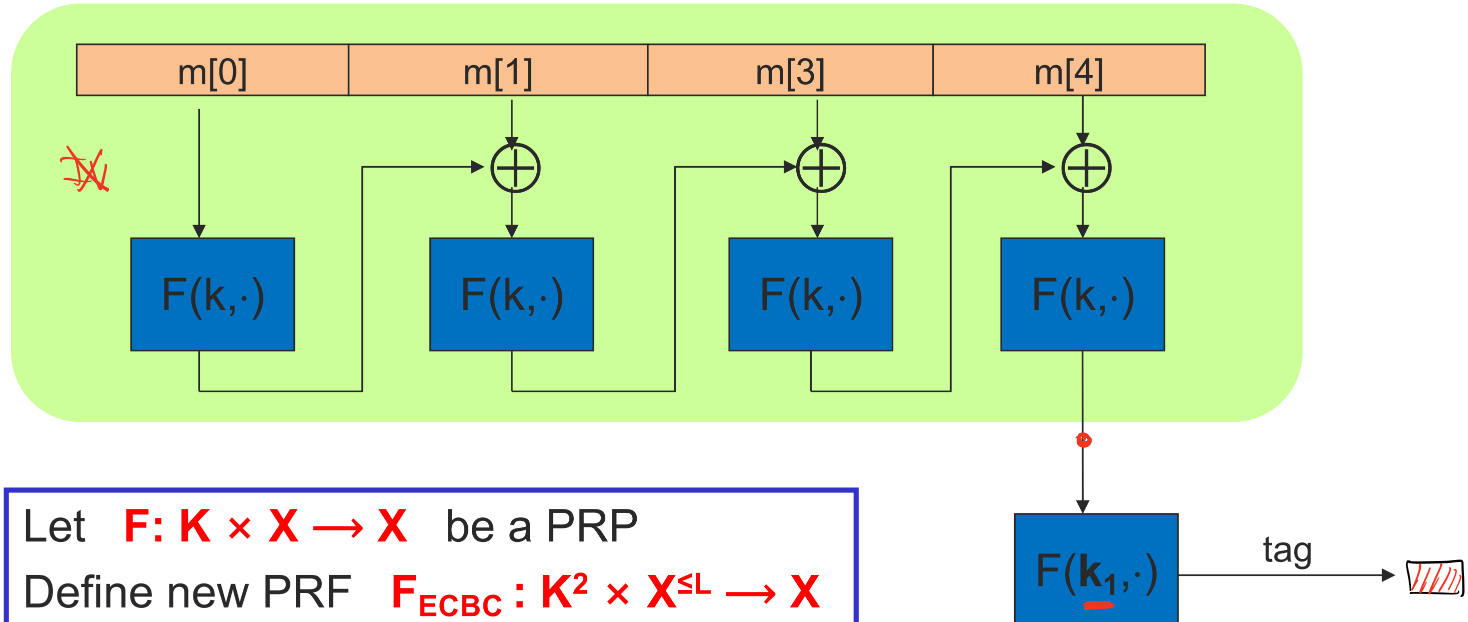
$C_3 \oplus \widetilde{m_4} = m_4$
$M^{\oplus} = (m_1, m_2, m_3, \widetilde{m_x}, m_5)$
$t^{\oplus} = \bar{t}$
$(m^{\oplus}, t^{\oplus})$ 가능X
1bit만 달라도 같은 m

1. IV문제 ⇒ IV를 제거(0으로)
가장 naive한 생각. but 바로 깨진다. 2. 이어붙이기 공격

- $(IV, tag=C_N)$ is a MAC tag with respect to message $M=(M_1,\ldots, M_N)$
- Is this CBC-MAC secure for arbitrary length N ?
  - Two reasons:

CBC-MAC – Practical Construction

raw CBC



Let **F: K × X ⟶ X** be a PRP

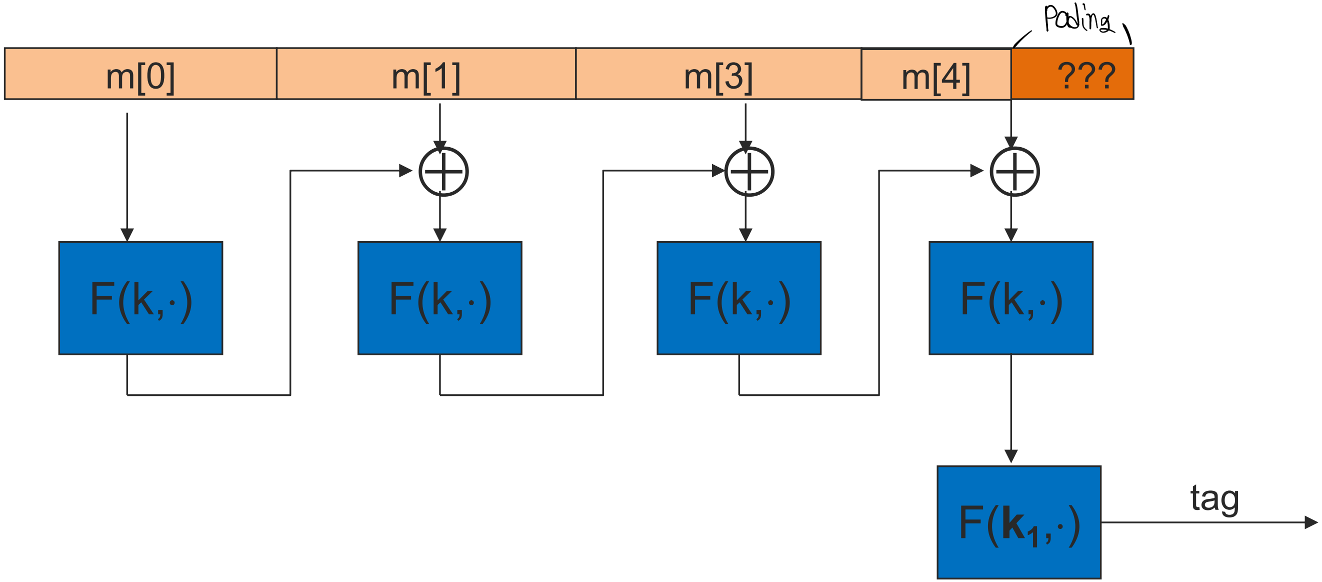
Define new PRF **F_{ECBC} : K^2 × $X^{\le L}$ ⟶ X**

key (k, k1)

- K = (k, k_1) should be shared in advance
- `tag` can be truncated with reasonable length
- What are differences between CBC encryption vs. CBC-MAC ?
- CBC-MAC is commonly used as AES-based MAC
  - AES-CCM encryption mode (used in IEEE 802.11i) CCA를 막기위해 Enc-then-MAC
    CBC or CTRmod CBC-MAC

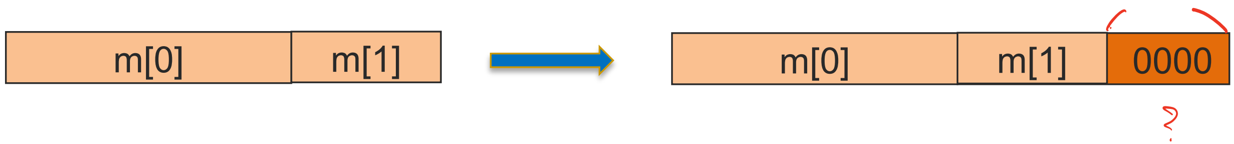
MAC padding

- What if msg. length is not multiple of block-size?



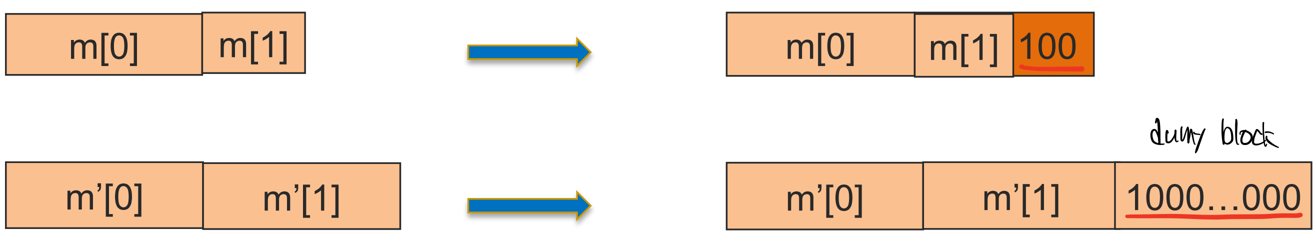
CBC-MAC padding

- Bad idea: pad m with 0's



  - Is the resulting MAC secure?

- ISO: pad with "1000…00". Add new dummy block if needed
  - The "1" indicates beginning of pad

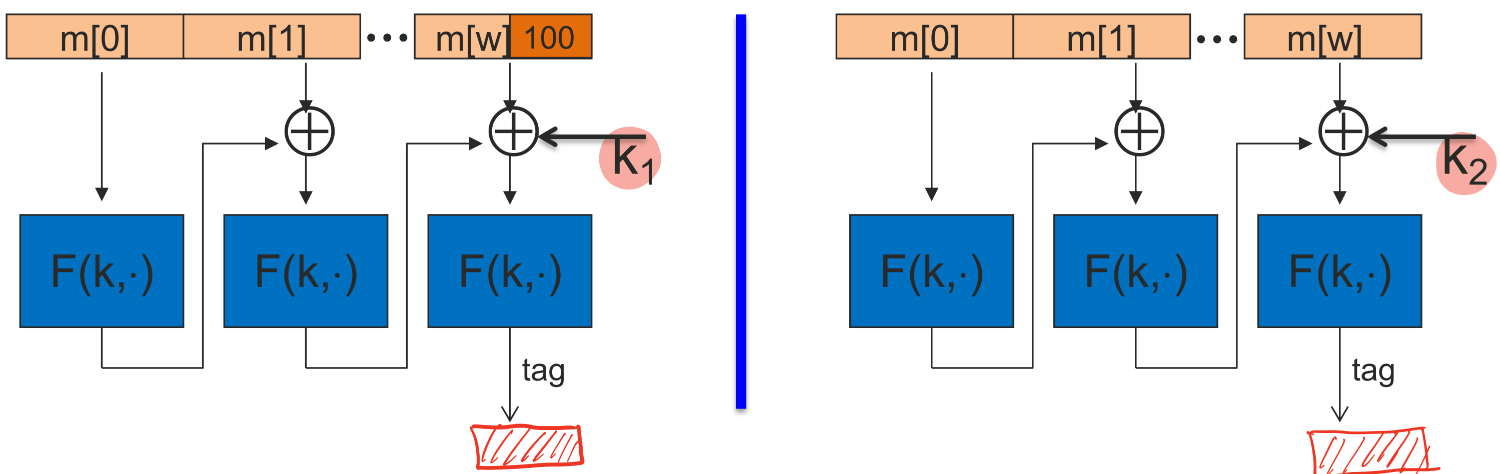


CMAC (NIST standard)

easy

(k_1, k_2) derived from K

- Variant of CBC-MAC where key = (k, k_1, k_2)
  - No final encryption step (extension attack thwarted by last keyed xor)
  - No dummy block (ambiguity resolved by use of k_1 or k_2)



1번더 암호화 X.

- What is the difference between CBC-MAC and CMAC?

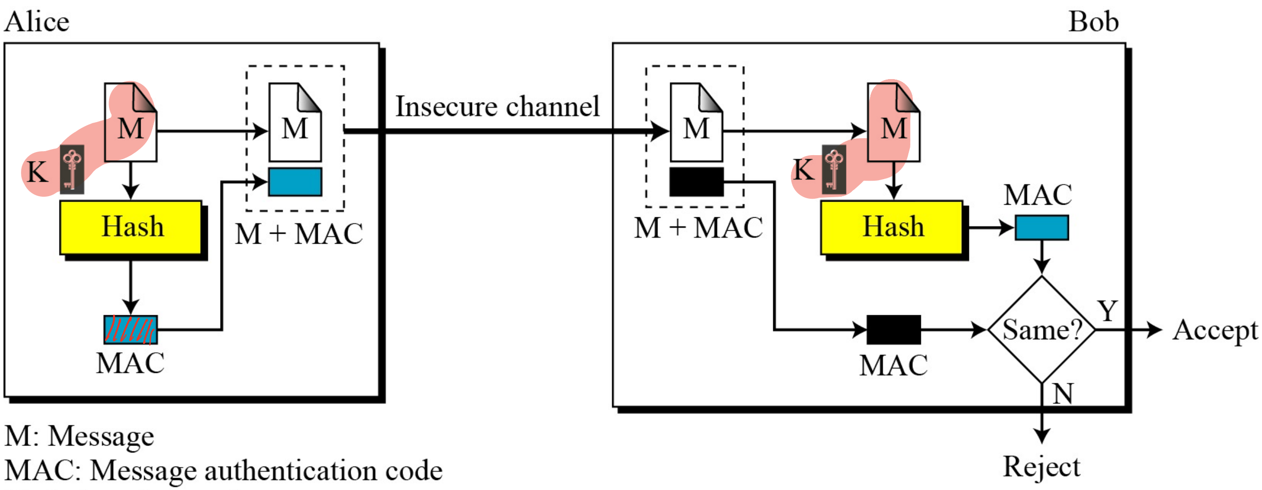
1. dumy block 생성 X
2. 1번더 암호화 과정 X.

HMAC – simple idea of using CRHF (1)

k

- Assume there exists a symmetric-key between Alice and Bob

K||M



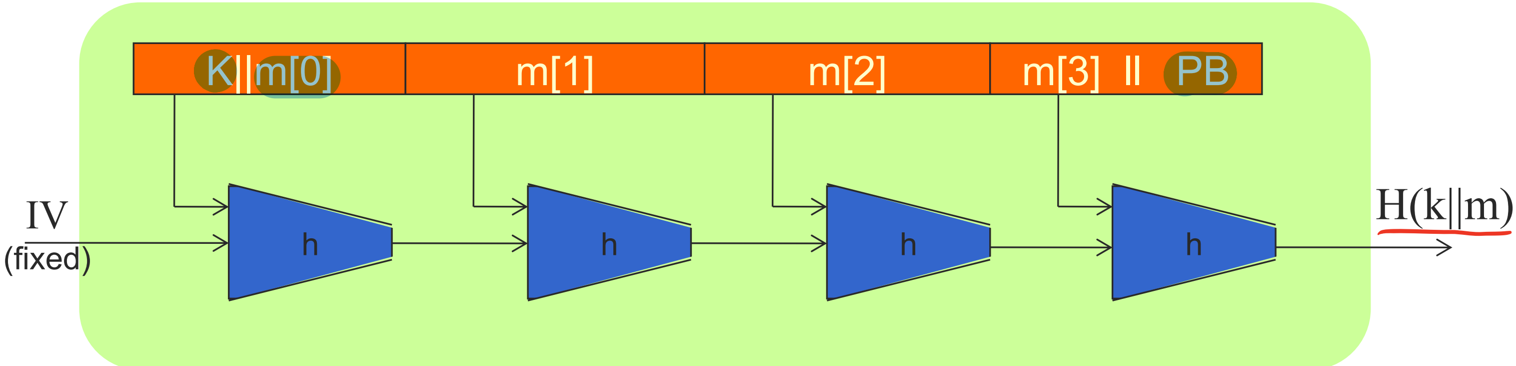
M: Message
MAC: Message authentication code
K: A shared secret key

사용이유 = 1bit라도 바뀌면 MAC가 달라진다.
uniform하게 분포에 key값만 넣어 HMAC구성
block 단위 처리속도가 빠르다.

- MAC is generated by H(K||M)
- When a hash function H is constructed using the Merkle-Damgard transform, the above method is not a secure MAC (why ?)

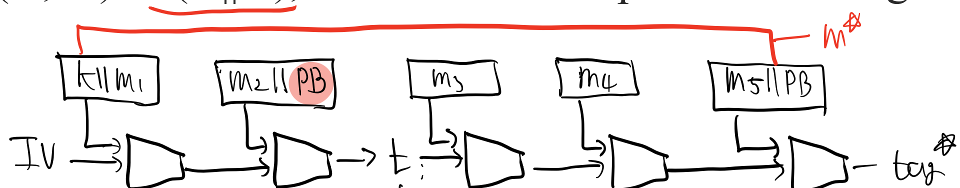
HMAC – simple idea of using CRHF (2)

- Merkle-Damgrd iterated construction



  - Given one Mac(K, m)=H(K||m), how can we compute another tag on a new message?

$(m_1, m_2) \rightarrow t_1$



(압축함수의 출력값으로 사용.

$\langle m^* = (m_1, m_2, m_3||PB, m_4, m_5),\ tag^* \rangle$

extension attack.

HMAC – Practical Construction

- How can we solve the above problem? → HMAC
  - NIST standard: most widely used MAC on the Internet
  - Building a HMAC from a hash function H

$$\text{HMAC: } Mac(k, m) = H(\, k \oplus opad \,\|\, \boxed{H(\, k \oplus ipad \,\|\, m \,)}\,)$$

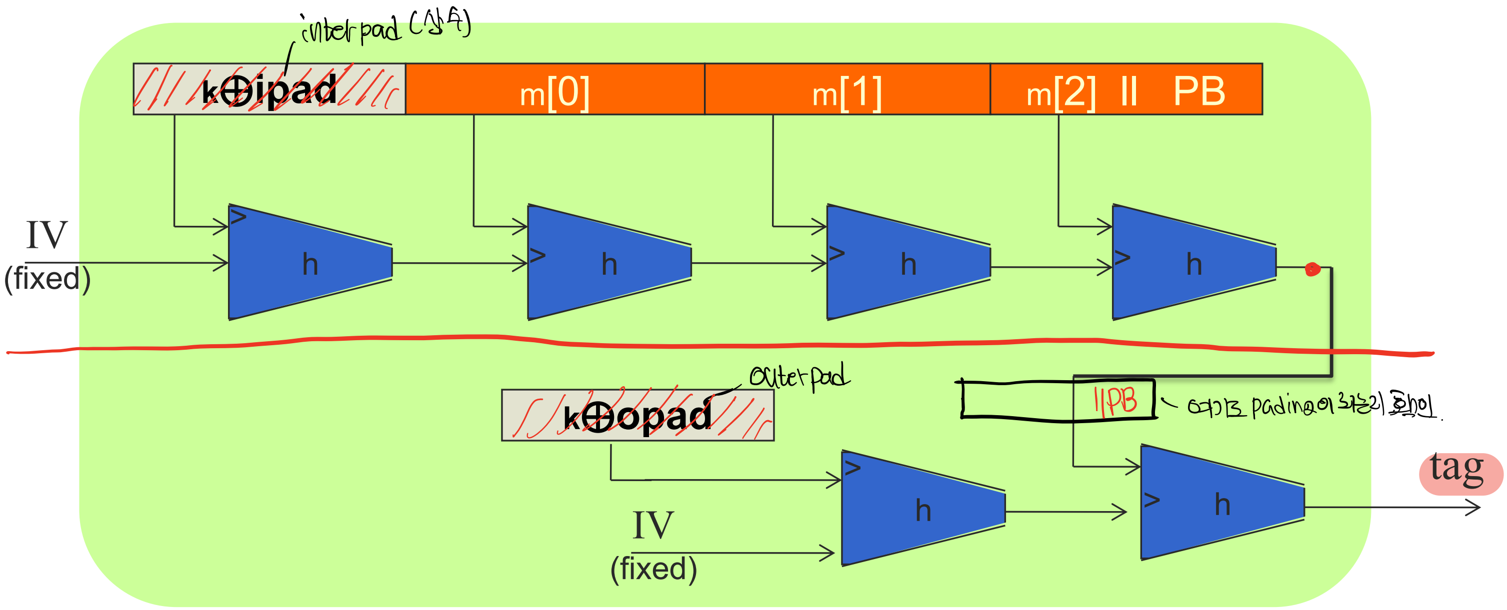
k , H

  - Fixed IV (코드상 X)
  - Using a single secret key K ipad , opad 코드는 봐보기
  - ipad = 0x36 (repeated), opad = 0x5C (repeated)

각주3갈 체크

HMAC in pictures

SHA-256



- ○ Two keys k_1 and k_2 are dependent, but …
- ○ In TLS: must support HMAC-SHA256-96

256bit를 96bit의 MAC

- Thm: If h is secure PRF and yields a secure fixed-length MAC, then HMAC is secure

Homework ~~3~~ 4

- https://github.com/intel/tinycrypt
  - HMAC (relying on SHA-256) code analysis (due date: 10/28)

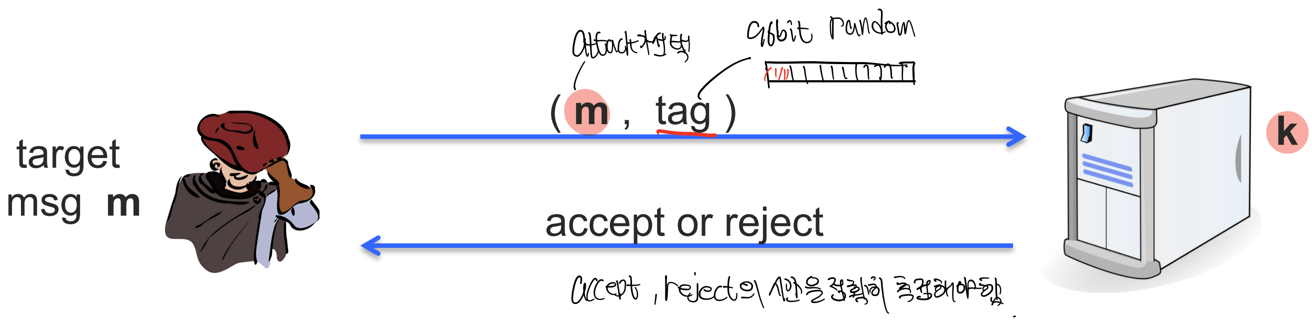
Verification Timing Attacks (1)

- Can be a generic attack for any MAC
- Given (M=message, T=tag)

```
Verify(Key, M, T):
    - generate MAC(Key, M)                ☆ 문제 발생
    - compare MAC(Key, M) == T (byte-by-byte comparison)
    - If equality does not hold, return 'error'
    - If equality holds, return 'authenticated'
```

- Problem: '==' implemented as a byte-by-byte comparison
  - Comparator returns false when first inequality found

Verification Timing Attacks (2)

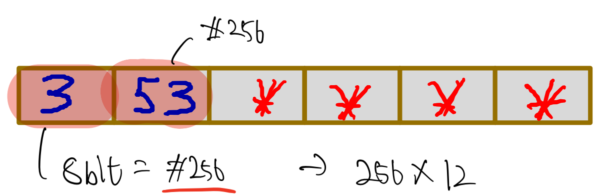


Timing attack: to compute tag for target message **m** do:

Step 1: Query server with random tag

Step 2: Loop over all possible first bytes and query server
stop when verification takes a little longer than in step 1

Step 3: repeat for all tag bytes until valid tag found



Defense

Make string comparator always take same time: 모든 byte를 확인후 return

constant time

```
def  Verify(key, M, T):
     tag = MAC(key, M)
     return MAC(key, tag) == MAC(key, T)
```

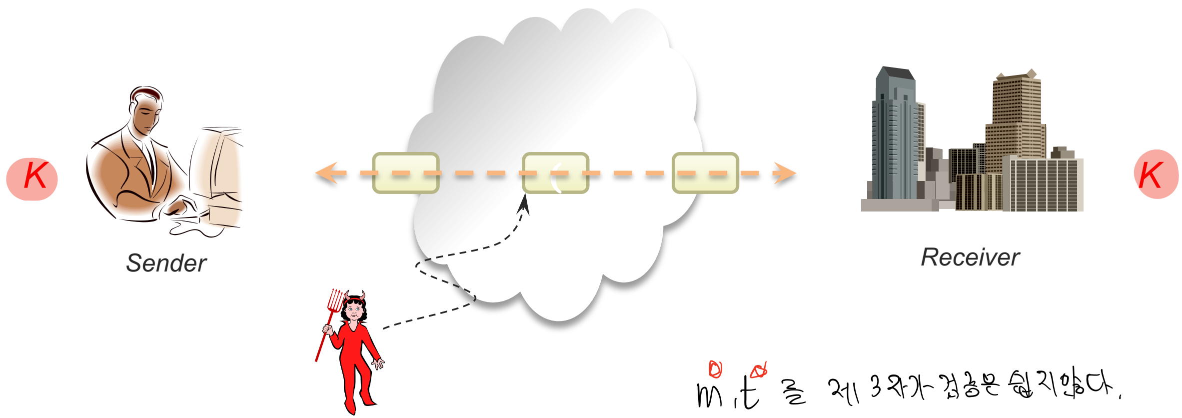
$\boxed{\qquad} == \boxed{\qquad}$

Attacker doesn't know values being compared

Lesson : Don't implement crypto yourself !

Limitation of MAC

- MAC is implemented in symmetric-key setting



m, t 를 제 3자가 검증은 쉽지 않다.

  - Later, if a legal dispute (about (m, t)) between sender and receiver happens, how can each entity convince other party that (m, t) is valid ?

- To solve the problem, need a new cryptographic primitive !!
  - Possible to provide 'public-verifiability'
  - Need 'digital signature' in public-key setting

integrity 보장, 검증가능.

Q & A